**Corso di Programmazione Web e Mobile**

**A.A. 2017-2018**

**Planet Defender Game**

Giovanni Sorgente [911248]

# Planet Defender

Una game app basata sul concetto di RTS (Real Time Strategy)

# Introduzione

Planet Defender, vuole essere una game app, basata su Angular 6 che si prefigge l’obiettivo di infoltire il parco applicazioni ludico fruibile sia da browser che da smartphone (Android).

La direttiva che sta alla base del progetto è quella di creare una sessione di gioco in modalità arcade, che permetta all’utente, in maniera del tutto gratuita di distrarsi e riempire quei “tempi morti” che spesso capitano durante la giornata.

Ad oggi, troviamo nei vari store sempre più applicazioni che presentano contenuti solitamente ridotti ma dall’obiettivo mirato al “divertimento” e svago temporaneo dell’utente.

Prendiamo ad esempio i casi più eclatanti che possono tornare alla mente, come Candy Crush oppure il meno famoso Flappy Bird; la richiesta è quella di avere un accesso veloce alla modalità di gioco, eventuali perks ed uno stretto ciclo di vita della partita che può essere ripetuta infinitamente rilassando l’utente che in quel frangente non dovrà impegnare troppo intensamente le proprie capacità intellettive ma potrà comunque trovare soddisfazione dall’attività svolta risolvendo i piccoli e semplici enigmi posti in essere dal gioco stesso.

A mio avviso la vastità del parco software presente online oggi permette sicuramente di usufruire di qualsiasi tipologia di applicazione si stia cercando; ovvero credo sia molto complicato trovare idee innovative che siano davvero originali e che possano essere poi proposte al grande pubblico.

Sono piuttosto convinto che la continua ricerca da parte dell’essere umano di qualcosa di diverso, di nuovo, di mai visto; porti poi, anche in questo ambito, al medesimo risultato del non rendere mai del tutto coperta una determinata tipologia di applicazioni.

Ecco che una piccola variante può così dal nulla creare una nuova versione che stimola di nuova la voglia di spendere del tempo da parte dell’utente nel superare quel determinato livello o sconfiggere quel determinato avversario.

Di seguito una breve descrizione del progetto che ho chiamato: **Planet Defender**.

* + Si tratta di un gioco fruibile sia da mobile che da web browser
  + E' previsto un login e relativa registrazione
  + L'utente (che visto l'ambito applicativo per comodità definisco giocatore), una volta effettuato il login, può decidere di cercare una partita in attesa oppure ospitarne una nuova ed attendere l'ingresso di un nuovo partecipante
  + L'iter di gioco si svolge in questo modo:
    - Esiste un'area virtuale (mappa) dove si sfidano due giocatori per volta, un "difensore" (giocatore in "casa") ed un giocatore "attaccante" (giocatore "fuori casa")
    - La mappa è definita da N tasselli in larghezza ed N in altezza
    - Ogni giocatore ha X carri armati a disposizione
    - Ogni carro armato ha un determinato numero di bonus vita
    - Prima di iniziare il turno:
      * ognuno dei giocatori posiziona le sue truppe sulla mappa. (premerà un pulsante per indicare di essere pronto per la partita)
      * inoltre il giocatore di casa posiziona N edifici da difendere durante la partita
    - Non vi saranno turni specifici per i giocatori, la fase di gioco si svolgerà infatti in tempo reale
    - Il giocatore può decidere di:
      * muovere il carro armato per un massimo (valore da definire) numero di tasselli
      * attaccare un carro armato avversario
      * se giocatore "attaccante" può decidere di attaccare anche gli edifici
    - Vincerà:
      * il giocatore "difensore" se riuscirà per primo ad eliminare la minaccia di attacco eliminando tutti i carri armati dell'"attaccante"
      * il giocatore "attaccante" se riuscirà per primo ad eliminare tutti gli edifici posizionati dal "difensore"

## Breve analisi dei requisiti

### Destinatari

Possiamo considerare l’applicazione come rivolta a chiunque. Questo per la bassissima curva necessaria per l’apprendimento delle funzionalità base proposte dall’interfaccia.

Ad ogni modo è probabile che il target si riferisca poi a persone di giovane età che solitamente si approcciano a questa tipologia di applicazioni ludiche.

Un’altro aspetto da considerare è che portando inoltre una grafica composta pressoché da icone ed immagini, è sicuramente necessario studiare nel dettaglio quanto ogni immagine scelta sia in grado di comunicare alla persona che utilizza l’applicazione; possiamo però essere certi del fatto che avendo poche azioni ed uno scopo univoco semplice e diretto che sarà molto semplice per l’utente perseguire l’obiettivo della partita.

### Modello di valore

Non è semplice stimare il valore dell’applicazione, una volta convertita in app mobile e portata sullo store, probabilmente con l’aggiunta di pubblicità potrebbe portare dei ricavi utili.

L’applicazione non da comunque un vero e proprio servizio, se non quello di poter intrattenere e rilassare il giocatore.

Rimane molto pulita e dall’idea semplice e lineare. L’unica leva possibile a meno di aggiungere ulteriori funzionalità future, sarà quella di ottenere delle buone statistiche e cercare di acquisire la testa della classifica acquisendo il maggior numero di vittorie possibili.

L’intrattenimento è certamente un altro valore aggiunto che come anticipato poco prima potrebbe essere la chiave di volta per creare una nuova applicazione, simile alle precedenti, ma che possa comunque offrire qualcosa di nuovo o di mai provato e che porti degli utenti sulla piattaforma per un periodo medio breve durante il quale si potrà cercare di massimizzare i ricavi attraverso gli introiti delle pubblicità.

I tempi di sviluppo sono certamente bassi e c’è inoltre da considerare, che valutando il modello di sviluppo adottato si potrà produrre con uno sforzo minimo un’applicazione similare in tempi ancora piu brevi.

### Flusso dei dati

La produzione di icone ed immagini qualitativamente soddisfacenti dovrebbe certamente portare in gioco l’introduzione nel team di sviluppo di un professionista del disegno di icone ed interfacce.

Ormai l’occhio abituato della maggior parte degli utenti è sempre più pretenzioso.

Trovare del materiale, come le icone dei carri armati o delle casette di buona qualità non è semplice e comunque è probabile che pur trovandole gratis o a prezzi convenienti, queste non corrisponderanno poi alle esigenze qualitative del prodotto.

Per quanto riguarda i dati grezzi, l’applicazione al momento tratta veramente pochi valori salvati staticamente in maniera duratura (utenti e statistiche utenti).

In genere uno degli aspetti più importanti è probabilmente quello di garantire la segretezza dei dati di accesso all’interno del proprio storage; ma possiamo dire che nono è questo il caso tenendo ben presente che non si tratta di dati sensibili è che in ogni caso riguardano poche statistiche di gioco che seppur venute in possesso di un malintenzionato non potrebbero ricondurre lo stesso alla vera identità dell’utente, né tantomento a sue informazioni strettamente personali (non salviamo nome utente e password).

### Aspetti tecnologici

Pensando all’implementazione dell’idea sopra descritta, probabilmente l’unica strada ad oggi percorribile, per avere un time to market accettabile è quella di utilizzare delle tecnologie multi piattaforma che permettano la flessibilità di distribuzione ed utilizzo da parte dell’utente.

Si pensi ad esempio alla vastità di versioni browser (desktop/mobile) ed alle versioni smartphone esistenti. Senza delle librerie e frameworks adeguati il tempo di sviluppo sarebbe così lungo che porterebbe a dei costi iniziali troppo alti per lanciare sul mercato una nuova idea applicativa.

Detto questo, si è deciso di propendere per tutte quelle tecnologie innovative che ad oggi la fanno da padrone in merito allo sviluppo web e mobile.

Di seguito vado ad elencare tutte le principali tecnologie scelte, tentando, per ognuna di darne una breve descrizione ed identificarne poi i principali punti di forza.

**Json**

Javascript Object Notation, è stato in grado in pochi anni di scalzare senza alcun dubbio Xml per gli scopi di serializzazione e passaggio dei dati tra servizi.

Possiamo dare alcune brevi caratteristiche:

* Schema less
* Poco verboso
* In grado di rappresentare qualsiasi struttra ad albero

Standard definito nel 1999 da Douglas Crockford che ancora oggi mantiene su GitHub un repository per un parser JSON scritto in javascript.

Non avendo schema, possiamo dire che è debolmente tipizzato, quindi la differenza in genere tra i tipi è data dal contenuto stesso della foglia (property dell’oggetto).

Ormai esistono serializzatori e deserializzatori di questo formato testuale che permette di adottarlo praticamente in qualsiasi situazione.

**Bootstrap**

Ad oggi è semplicemente lo standard de facto per la realizzazione degli scheletri di pagine web e mobile.

Si tratta di una suite di librerie css e javascript che permette di generare in maniera molto semplice e rapida un layout responsivi senza doversi preoccupare delle risoluzioni dell’ambiente ospitante.

Per layout responsive intendiamo una grafica che sia in grado di adattarsi dinamicamente alle principali risoluzioni dei dispositivi ad oggi presenti sul mercato.

L’avvento di una libreria di questo tipo accelera considerevolmente i tempi di sviluppo permettendo, cosa di non poco conto anche a sviluppatori puri, senza un’anima dedita al disegno di produrre interfacce efficaci e valide.

Javascript

Linguaggio di scripting creato da Brendan Eich nel 1995; a mio parere uno dei più flessibili e comodi linguaggi presenti ad oggi per la creazione di applicazioni di vario genere.

Pur essendo nato principalmente ai tempi con l’obiettivo di costruire script dinamici nei primi browser nascenti, oggi si afferma sempre più come linguaggio utilizzato anche per lo sviluppo di servizi con il noto NodeJs.

Standard EcmaScript, si afferma inizialmente per i primi browser dell’epoca come Netscape.

Si ferma per anni a partire dal 2008 alla versione 5, che già comprendeva un buon numero di funzionalità.

Il 2015, anno di svolta, in cui viene rilasciata come standard la versione 6 che rispetto alla precedente versione porta innovazioni al linguaggio tali da renderlo sempre più vicino ad un vero e proprio linguaggio ad oggetti.

Abbiamo infatti nuove keywors come class (purtroppo non ancora i modificatori di visibilità come private), const, let (utilissimo per ridurre lo scope delle variabili), iteratori, ecc…

Di recente nel 2017, dato il modello asincrono con cui spesso deve confrontarsi un’app scritta in javascript, viene introdotto anche il costrutto async/await.

***Riferimenti***

***https://it.wikipedia.org/wiki/ECMAScript***

Typescript

Definito in genere come un “super set” di istruzioni costruito sopra Javascript, prima versione risalente al 2012 distribuita da Microsoft, seguita dal capo progetto Anders Hejlsberg; ingegnere danese che ai tempi della Borland ha sviluppato il famoso compilatore Pascal (Turbo Pascal).

Interessante l’adozione dello stesso da parte di google per le nuove versioni di Angular, anche se la scelta è più che comprensibile dato l’alto livello di controllo dato dal linguaggio.

In typescript vengono infatti introdotti alcuni dei principali costrutti dei linguaggi orientati ad oggetti, cercando in un certo senso, di sopperire alle mancanze che ad oggi Javascript ancora possiede.

Tra queste, possiamo citare:

* Tipi delle variabili assegnabili a compile time, quindi esiste la possibilità di effettuare un check statico dal compilatore per verificare la compatibilità dei tipi assegnati
* Possibilità di utilizzare il polimorfismo ad un più alto livello attraverso l’ausilio delle interfacce
* Modificatori di proprietà di classe: private, public, protected

Ricordiamo inoltre che Typescript arriva prima di ES6, ragion per cui lo ha popolare in maniera molto rapida, offrendo ciò che nativamente ancora non era possibile scrivere.

Dato l’utilizzo fatto durante lo sviluppo dell’applicazione, mi sembra doveroso precisare che quanto detto offre un vantaggio estremo per lo sviluppatore che come spesso accade con javascript, si accorge di qualche problema nel proprio codice solo in fase di esecuzione.

***Riferimenti***

***https://www.typescriptlang.org/***

Angular 6/Angular-cli

Angular ad oggi in versione 6, nasce nel 2010 con una prima versione scritta interamente in javascript che si basava fortemente sul pattern MVC.

L’idea alla base del framework è quella di poter definire modelli dati, viste e controllori ben disaccoppiati tra loro.

Nasce principalmente per la creazione di SPA (single page application) che si muovono sul filone logico di scaricare l’intera applicazione web al primo accesso e poi scambiare esclusivamente dati di business con il server.

Una delle caratteristiche principali, per cui probabilmente si ricorda angular, è la cosiddetta modalità del two way binding che permette di aggiornare in maniera rapida l’interfaccia utente modificando esclusivamente i dati a disposizione del controllore.

Un’altra importante funzionalità che angular porta gratuitamente nelle tasche dello sviluppatore è il pattern IoC (Inversion of control), difatti, ad ogni controllore è possibile “iniettare” una determinata istanza di un nostro servizio per poter permettere il riutilizzo di codice e l’ausilio dell’implementazione desiderata del servizio richiesto.

Dalla versione 2 in poi, nata nel 2016, viene adottato Typescript per la scrittura dello stesso tipo di applicazioni SPA.

Come detto in precedenza oggi siamo alla versione 6, le caratteristiche del framework sono molteplici, però possiamo riassumerle affermando che in genere l’implementazione di una data funzionalità attraverso servizi e componenti.

In genere un servizio è definito da una classe “decorata” dall’attributo @Injectable mentre un componente viceversa dall’attributo @Component. Il primo definisce macro funzionalità a livello applicativo che vengono condivise tra i vari componenti, metre il secondo scende nello specifico ed implementa la porzione di interfaccia utilizzata dall’applicazione.

Ma qual’è inoltre il grosso vantaggio che un’azienda dovrebbe considerare nel seguire

un approccio di questo tipo?

Il principale vantaggio da evidenziare nell’adozione di questo modello di sviluppo, sta sicuramente nel fatto che un team di sviluppo è in grado di separare e fornire più agilmente il lavoro ai componenti del gruppo, dato che il framework stesso nasce con l’idea di garantire questa separazione di ambiti.

Garantisce sicuramente uno standard nella modalità di lavoro, questo permette certamente l’inserimento di una nuova risorsa in tempi più brevi.

***Riferimenti***

[***https://angular.io/***](https://angular.io/)

WebSocket/Socket.io

MongoDB

La scelta di MongoDB, ad oggi uno tra i più conosciuti DBMS non relazionali, nasce dall’esigenza di poter gestire una grossa mole di dati senza uno schema database ben preciso.

Basti pensare ad esempio alla necessità di salvare le statistiche utente che, a fronte di un grosso numero di registrazioni, potrebbero richiedere il salvataggio di un gran numero di statistiche ed informazioni da salvare.

Un futuro sviluppo dell’applicazione potrebbe ad esempio dare la possibilità di visionare un match in corso, fornendo agli spettatori funzionalità di commentare la partita o marchiare con una spunta di gradimento le mosse dei giocatori.

Tutto questo, visto in larga scala, potrebbe sicuramente essere reso più performante denormalizzando le strutture dati in essere e gestendo il tutto senza relazioni tra gli oggetti.

***Riferimenti***

***https://www.mongodb.com***

Node.js/Javascript

Nodejs ….

Snippets e modalità di utilizzo

<creazione api, modello codice backend, … >

***Riferimenti***

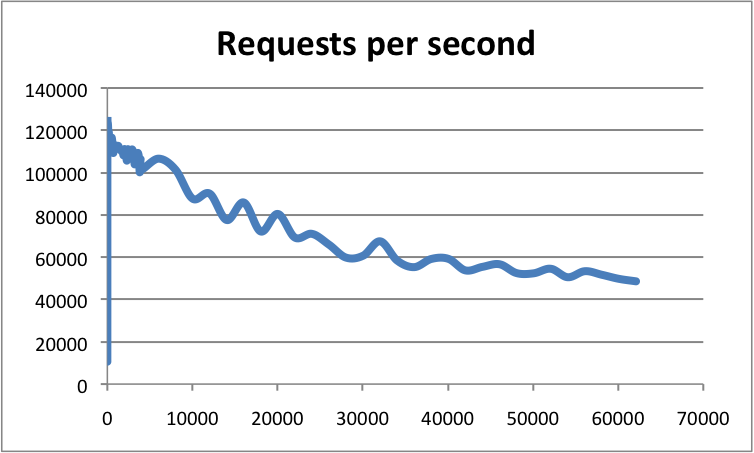
*https://nodejs.org/it/*

Redis

Redis fa parte della categoria di prodotti cosiddetti datastore in memory. Sviluppato inizialmente da un italiano, Salvatore Sanfilippo che decide di mantenere il progetto open source (<https://github.com/antirez/redis>), diventa nel breve periodo una community ben più ampia e che oggi probabilmente deve la propria popolarità al gruppo Redis Labs che ha pensato alla commercializzazione di servizi per distribuire il prodotto stesso.

L’adozione di Redis nasce dall’esigenza di persistere in maniera volatile dei dati di gioco in maniera rapida e veloce.

A questo punto, si è scelto di proseguire con la tecnologia in oggetto data la comprovata rapidità nel gestire le richieste, documentata anche dai benchmarks forniti direttamente sul sito del prodotto (<https://redis.io/topics/benchmarks>).



***Riferimenti***

[*https://github.com/noderedis/node\_redis*](https://github.com/noderedis/node_redis)

*https://redis.io/*

### PaaS – Sistema di distribuzione ed hosting della piattaforma

Arrivati a questo punto, c’è la necessità di capire come distribuire e rendere disponibile in tutta internet il nostro servizio.

Chiaramente, date le tecnologie utilizzate, abbiamo bisogno di un ecosistema architetturale abbastanza complesso, che metta a disposizione tutta una serie di servizi (database, compiler node, memory cache, ecc…) e che soprattutto sia in grado di configurare i vari prodotti in uso senza rendere troppo complesse le interoperabilità tra i servizi scelti.

L’unica soluzione al momento disponibile è adottare un servizio PaaS (Product as a Service) che pensi al posto nostro a configurare e rendere operativo l’intero parco applicativo.

PaaS

Pubblicazione del proprio prodotto all'interno di un'applicazione hosting che si preoccuperà di distribuire e rendere fruibile da parte del nostro cliente quanto sviluppato. A questo punto non ci preoccupiamo dell'infrastruttura di rete, delle configurazioni di bilanciamento tra server e di qualsiasi ammenicolo legato a macchine virtuali o altro.

Heroku

Tra le varie opportunità sul mercato, la soluzione proposta da Heroku sembra la più appetibile in termini di costi/benefici.

Si basa su sistemi UNIX.

Struttura:

**Routers**: gestione del routing dei servizi, redirezionano le chiamate al corretto gestore. Praticamente è uno zoccolo scritto in Erlang (Hermes) che fa da bridge tra la nostra applicazione e il dyno.

**Dynos**: potremmo vederla come una macchina virtuale o un virtual server che esegue il nostro codice. Generalmente un dyno viene fornito con 512RAM . Le applicazioni vengono appunto principalmente fatte pagare per dyno/ora con un dettaglio al secondo.

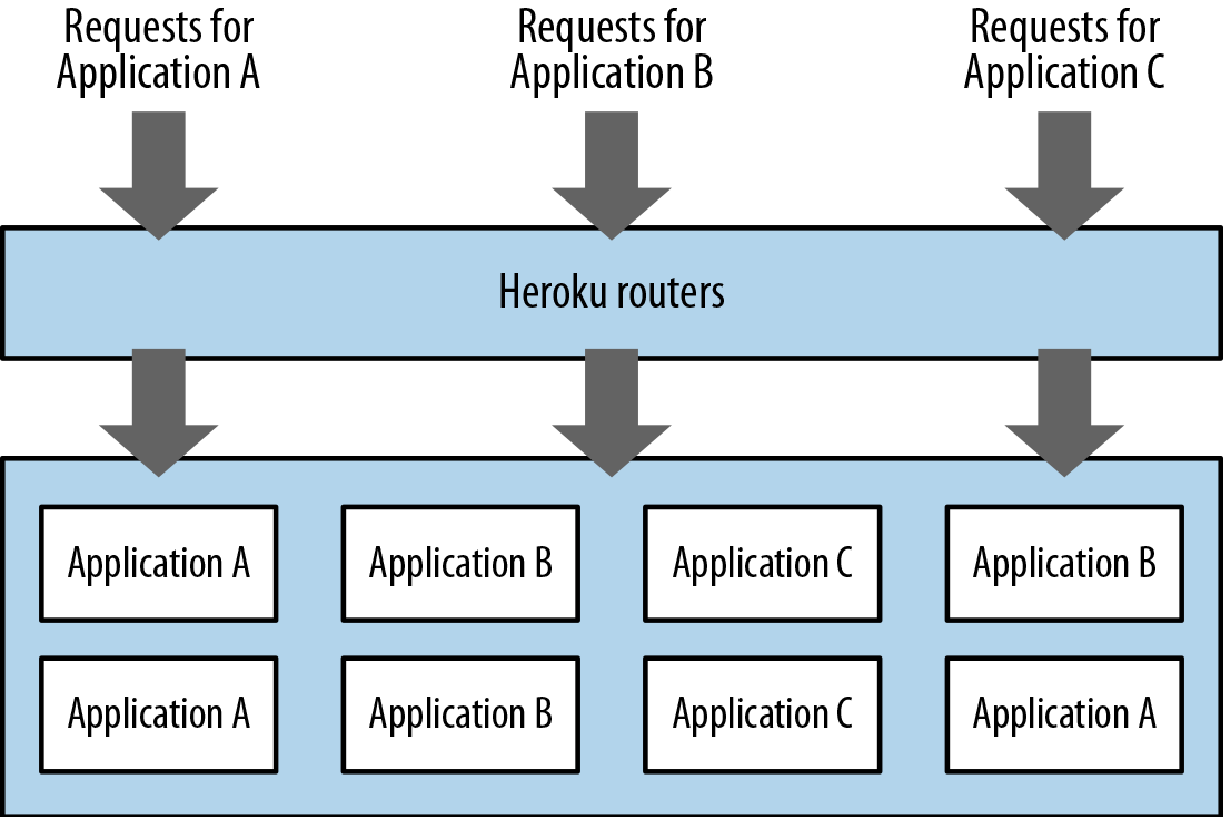
Chiaramente più è maggiore l’investimento economico, più è performante il dyno.

Più dyno si hanno a disposizione, più richieste concorrenti è possibile gestire.

Ogni dyno viene riavviato ogni 24 ore.

**Componenti aggiuntivi**: tutto ciò che viene fornito come componente software (servizio) da pagare a parte (vedi MongoDB, Redis, ecc...).

Il funzionamento generale prevede il commit attraverso git del software, questo una volta ricevuto da Heroku, provvede ad incapsularlo in un codice comprensibile da Heroku stesso in gergo definito come “slug”. Il sistema di routing invece, rende trasparente allo sviluppatore l’astrazione tra il linguaggio e l’application server (Tomcat, Unicorn, ...) che si occupa del dispaccio dei messaggi http verso i client. Il risultato è quello di permettere allo sviluppatore di scrivere il codice con il linguaggio per lui più consono e conosciuto, delegando poi alla piattaforma il compito di distribuirlo e renderlo operativo.



Gestione fault tolerance automatica tra i dyno.

Il grosso vantaggio inoltre offerto da un sistema di questo tipo, è la scalabilità offerta dai “contenitori” (dyno) che ospitano il codice dello sviluppatore; in poche parole, è sufficiente investire più denaro sulla piattaforma per ottenere più potenza di calcolo e memoria da garantire al proprio software. Come se non bastasse, a fronte di un maggior numero di richieste in ingresso, l’acquisto di più contenitori permette di portare il proprio applicativo verso un’architettura a microservizi, che se scritto opportunamente, potrà essere replicato per poter far fronte al bisogno di gestire le crescenti richieste verso il proprio sistema.

***Riferimenti***

*https://www.heroku.com/*

# Interfacce

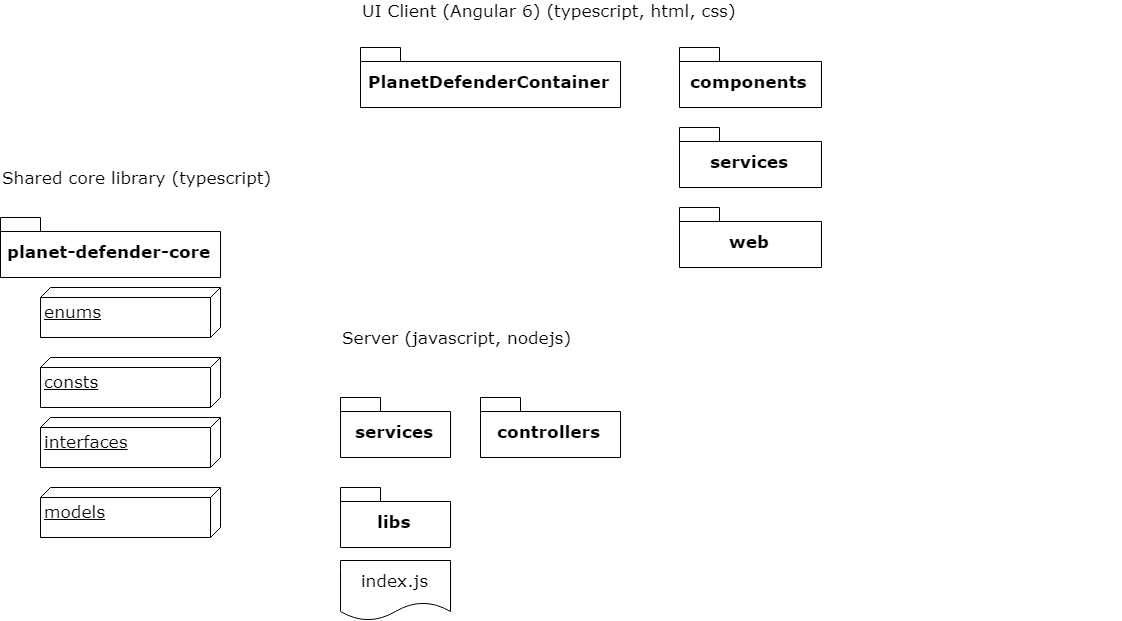
Di seguito andiamo ad illustrare le interfacce che si intende realizzare per il progetto in questione.

|  |  |
| --- | --- |
|  | **Login**  In caso di primo accesso o sessione scaduta, l’utente deve autenticarsi attraverso questa schermata.  La form prevede inoltre un pulsante per la registrazione di un nuovo utente. |
|  | **Schermata di gioco**  La scheramata viene divisa in quadrati di dimensione fissa. Ognuno di questi può contenere degli oggetti di gioco.  L’utente sarà in grado di selezionare l’elemento voluto e spostarlo nella cella desiderata.  Un’altra azione disponibile sarà l’attacco che permetterà l’eliminazione degli oggetti in mappa dell’avversario. |
|  | **Registrazione**  Nel caso in cui un nuovo utente voglia utilizzare l’applicazione, verrà messa a disposizione questa schermata di registrazione che permetterà la creazione dell’utente sulla base della specifica di un nome utente ed una password. |
|  | **Menu**  Il menu permetterà il passaggio tra le varie schermate di gioco.   * Host game/Nuova partita: renderà disponibile online una nuova area di gioco dove due sfidanti potranno confrontarsi * Find game/Unisciti ad un partita: cerca una partita esistente ed introduce l’utente all’interno della partita in gioco * Statistics/Statistiche utente: sono le statistiche del giocatore “loggato”. Permette di sapere quante partite ha vinto o perso e quanto unità o edifici ha perso durante le battaglie |
|  |  |

# Architettura

## Diagramma dell'ordine gerarchico delle risorse

L’architettura dell’intera applicazione è abbastanza vasta. L’interezza del codice è stato scritto in linguaggio di scripting sia javascript che typescript.

Figura 1: Architettura delle applicazioni

Andando per ordine, possiamo identificare tre macro sezioni all’interno dell’intero progetto:

* **planet-defender-core:** libreria applicativa scritta interamente in typescript. Contiene tutto il codice comune tra il server ed il client. Questa libreria viene compilata in attraverso il tool “tsc” eseguibile e compilatore per typescript. La componente npm che si occupa di tutta la fase di impacchettamento e distribuzione si chiama ng-packgr. Questo pacchetto nasce dall’idea di condividere tutto il codice necessario tra l’applicazione front-end e back-end.

Incapsula inoltre alcune logiche di business, quali il reperirmento e la gestione degli oggetti in mappa.

Come evidenzia lo schema, inoltre, è evidente che questo condivida per la maggiore tutti i datamodel utili sia alla costruzione dell’interfaccia che alla gestione dell’area di gioco lato server.

* **PlanetDefenderContainer**: questa è l’applicazione front-end scritta in Angular 6.

Come già evidenziato prima, basa la propria logica di business sui datamodel definiti nella precedente libreria.

Si caratterizza di tre fondamentali risorse:

* + *components:* contiene tutte quelle viste che servono per gestire l’applicazione (menu, login, ecc…) e per l’intera area di gioco (carri armati, edifici, …)
  + *services*: sono i servizi applicativi, molti dei quali ereditano dalle interfacce definite nella libreria core. Si preoccupano di gestire tutta la logica di gioco e di interpellare il server qualora serva autenticarsi o interagire con le logiche della partita.
  + *web*: è il collettore delle precedenti librerie, referenzia entrambe ed espone una sola vista (quindi un solo index.html ed un solo componente) che amalgama tutte quelle componenti definite nelle altre due librerie

Questa scelta implementativa, nasce dalla volontà di volersi rendere compatibili anche con ulteriori versioni di sviluppo.

La possibilità di avere delle librerie di componenti e servizi disaccoppiate tra loro ed altamente coese, permetterebbe la realizzazione di una successiva implementazione dell’applicazione questa volta su mobile, riutilizzando pressoché tutto il codice già sviluppato.

* **Server:** l’ultima parte, non meno importante, è caratterizzata dal codice del servizio, che come già descritto prima, viene ospitato direttamente su Heroku. L’applicazione fa largo uso della libreria core, che viene compilata per javascritpt.

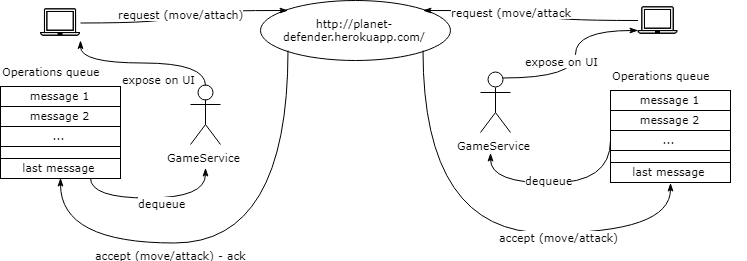
Anche qui troviamo la separazione tra controllori e servizi. I primi si preoccupano di interagire con le risorse primitive dell’ecosistema architetturale (mongodb, redis) oppure di astrarre tutte le logiche di business dell’applicazione (registrazione/login, nuova partita, …).

I controllori invece, espongono meramente verso l’esterno dei punti di confine che permettono l’accesso alle risorse appena elencate dall’esterno.

Questi sono generalmente agnostici dell’intera logica di lavoro e si limitano ad interrogare quelli che sono i servizi per poter completare le proprie azioni.

## Descrizione delle risorse

L’architettura principale che viene implementata per poter garantire l’interoperabilità tra le parti mantenendo però la consistenza della partita e delle operazioni svolte dai giocatori, viene descritta nel diagramma appena sopra.

Figura 3: Architettura utilizzata.

Come si può evincere dalla figura, ogni azione effettuata da un utente, si ripercuote sull’interfaccia che prima di agire e rendere effettiva quell’attività, decide di comunicarla al server.

Il server si prodiga per gestire la richiesta appena inoltrata, ne gestisce eventuali collisioni e se questa operazione viene decretata come eseguibile, attraverso una notifica web socket, il server informa il client dell’esito della sua richiesta.

L’attività ciclica appena descritta non può essere rappresentata in maniera sincrona, questo perché l’applicazione non vuole essere un gioco basato su turni da attendere, vuole invece essere dinamico e parallelo per quanto possibile nell’esecuzione dei suddetti comandi.

# Codice

*Frammenti del codice più significativo*

## Node.js

## Autenticazione

processWithAuthentication : (businessLogicCallback) => {

return function(request, response, next) {

// do authentication stuffs

var token = request.body.token || request.query.token || request.headers['x-access-token'];

// verifies secret and checks exp

jwt.verify(token, serviceConst.AUTH\_TOKEN\_SECRET, function(err, decoded) {

if (err) {

response.send(401);

return response.json({ success: false, message: 'Failed to authenticate token.' });

}

});

if (businessLogicCallback) {

businessLogicCallback(request, response, next);

}

return null;

}

}

L’autenticazione basata su token JWT (javascript web token), viene verificata ad ogni richieta. Il codice recupera il token di autenticazione dall’header (x-access-token) e ne verifica la validità confrontando l’hash passato dal client con quello prodotto utilizzando la chiave privata del server AUTH\_TOKEN\_SECRET.

## Web Socket (push messaggi verso il client)

## /\*\*

\* Send message to the specified channel

\*/

sendMessage(channelId, eventName, message) {

this.service.of(‘/’ + channelId, eventName, message);

}

Invio di messaggi verso il client sul namespace specificato. In questo caso, via protocollo web socket, il server manda forzatamente al client il messaggio del comando da eseguire.

## CSS

## Animazione carriarmati

-webkit-transition: left 0.7s linear, top 0.7s linear;

-moz-transition: left 0.7s linear, top 0.7s linear;

-o-transition: left 0.7s linear, top 0.7s linear;

transition: left 0.7s linear, top 0.7s linear;

animation-iteration-count: infinite;

# Conclusioni

In conclusione possiamo dire che il progetto seppur complesso dal punto di vista architetturale, porterà ad un ottimo prodotto essendo stato implementato su di una piattaforma che offre scalabilità e performance.

L’ulteriore grosso vantaggio che si rileva dall’approccio adottato è la portabilità di quanto scritto; a fronte di codice identico e sviluppabile su un unico repository, possiamo ottenere la stessa applicazione distribuita su più piattaforme.

Al momento quanto sviluppato rimane come prova di concetto in quanto la commercializzazione di software di questo tipo richiedono certamente un impegno ulteriore nel delineare una grafica molto più accattivante.

Abbiamo comunque dimostrato che attraverso lo sviluppo di codice completamente basato su javascript è possibile ottenere del software che ricopre interamente gli attori di una piattaforma client server.